Patterns in datatype-generic programming (extended abstract)

Jeremy Gibbons
Oxford University Computing Laboratory

February 2004

1 Introduction

Generic programming [14, 10] is a matter of making programs more adaptable by making them more general. In particular, it consists of allowing a wider variety of entities as parameters than is available in more traditional programming languages.

The most popular instantiation of generic programming today is through the C++ Standard Template Library (STL) of container classes and generic algorithms. *Datatype-generic programming* (DGP) is another instantiation of the idea of generic programming. DGP allows programs to be parameterized by a *datatype* or *type functor*. DGP stands and builds on the formal foundations of category theory and the *Algebra of Programming* movement [3, 2, 5], and the language technology of Generic Haskell [11, 6].

We argue that DGP is sufficient to express essentially all the genericity found in the STL. In particular, we claim that various programming idioms that can at present only be expressed informally as *design patterns* [8] could be captured formally as datatype-generic programs. Moreover, because DGP is a precisely-defined notion with a rigorous mathematical foundation, in contrast to generic programming in general and the C++ template mechanism in particular, this observation offers the prospect of better static checking of and a greater ability to reason about generic programs than is possible with other approaches.

This paper describes the work to be undertaken in the UK EPSRC-funded project *Datatype Generic Programming*, starting October 2003. The work described in this paper will constitute about a third of that project; a second strand, coordinated by Roland Backhouse at Nottingham, is looking at more of the underlying theory, including logical relations for modular specifications, higher-order naturality properties, and termination through well-foundedness; the remainder of the project consists of an integrative case study. This is an extended abstract of a longer paper [9].

2 Datatype genericity

The essence of DGP is the parametrization of values (for example, of functions) by a *datatype*. Consider for example the two parametrically polymorphic programs for mapping over lists and over trees. Both of these programs are already quite generic, in the sense that a single piece of code captures many different specific instances. However, even though their types are similar, their implementations are not. A DGP language would allow their common features to be captured in a single definition parametrized by a datatype (lists or trees).

The STL permits the expression of operations like map, but does not permit a program to *exploit* the shape of the data on which it operates. For example, a datatype-generic program to encode a data structure as a bit string [13] uses the *shape* of the input to determine the *value* of the output. A more sophisticated example involves Huet's 'Zipper' [12]; different shapes of input yield different *shapes* of output. Neither of these examples are possible with the STL.

The full version of the paper [9] explains why DGP is not the same thing as *meta-programming*, various kinds of *polymorphism*, *dependently typed programming*, or Generic Haskell [6].

3 Patterns of software

A *design pattern* 'systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems' [8]. The intention is to capture best practice and experience in software design. It could be argued that many of the standard patterns are idioms for mimicking DGP in languages that do not properly support such a feature. Because of the lack of proper language support, a pattern can generally do no better than to motivate, describe and exemplify an idiom: it can refer indirectly to the idiom, but not present the idiom directly as a formal construction. For example, the ITERATOR pattern shows how an algorithm that traverses the elements of a collection type can be decoupled from the collection itself, and so can work with new and unforeseen collection types; but for each such collection type an appropriate new ITERATOR class must be written. (The programmer may be assisted by the library, as in Java, or the language, as in C♯, but still has to write something for each new collection type.) A DGP language would allow a single datatype-generic program directly applicable to an arbitrary collection type.

4 Future plans

In the short term, we hope to prototype a datatype-generic collection library (perhaps as a refinement of Okasaki's Edison library [17]). But the real purpose of the project will be to generalize theories of program calculation such as Bird and de Moor's relational 'algebra of programming' [5], to make it more applicable to deriving the kinds of programs that users of the STL write. This will link with Backhouse's strand of the DGP project, which is looking at more theoretical aspects of datatype genericity.

More tangentially, we have been intrigued by similarities between some of the more esoteric techniques for template meta-programming [7, 1] and some surprising possibilities for computing with type classes in Haskell [16, 15, 4].

References

[1] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[2] R. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T. Voermans, and J. van der Woude. Polynomial relators. In *AMAST*, 1992.

[3] R. Backhouse, P. de Bruin, G. Malcolm, T. Voermans, and J. van der Woude. Relational catamorphisms. In *Working Conference on Constructing Programs from Specifications*, 1991.

[4] R. Backhouse and J. Gibbons. Programming with type classes. Presentation at WG2.1#55, Bolivia, January 2001.

[5] R. S. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[6] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. The Generic Haskell user's guide. Technical report, Universiteit Utrecht, 2001.

[7] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] J. Gibbons. Patterns in datatype-generic programming. In J. Striegnitz, editor, *DPCOOL*, 2003.

[10] J. Gibbons and J. Jeuring, editors. *Generic Programming*. Kluwer Academic Publishers, 2003.

[11] R. Hinze. Polytypic values possess polykinded types. *Sci. Comput. Prog.*, 43, 2002.

[12] G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, Sept. 1997.

[13] P. Jansson and J. Jeuring. Polytypic data conversion programs. *Sci. Comput. Prog.*, 43, 2002.

[14] M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors. *Generic Programming*. Springer-Verlag, 2000.

[15] C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Prog.*, 12, 2002.

[16] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. Functional logic overloading. In *POPL*, 2002.

[17] C. Okasaki. An overview of Edison. Haskell Workshop, 2000.